Lab 4. AI Agents

(Duration: 2h)

1. Introduction

This lab introduces **AI agents** as autonomous systems that perceive their environment, reason about what to do, and act by invoking tools or producing responses. You will implement an agent from first principles using LiteLLM, Pydantic, asyncio, and tenacity. The focus is on core agent loops, tool calling, asynchronous execution, robust error handling, schema validation, and production-ready practices like logging and monitoring.

2. Learning Outcomes

By the end of the session, you should be able to:

- Understand the fundamental architecture of AI agents.
- Implement agent reasoning loops without relying on high-level frameworks.
- Design and implement tool-calling mechanisms.
- Handle asynchronous operations in agent systems.
- Implement robust error handling and retry mechanisms.
- Validate agent inputs and outputs using structured schemas.
- Build production-ready agents with proper logging and monitoring.

3. Preparation

Before the session, ensure that you have completed the following:

1. **Downloaded the repository:** Clone or download the lab template from https://github.com/intelligent-process-automation-IPA/ai-agents.
2. **Setup API credentials:** Use your Groq API key from Lab 2 for the language model. Add the following to your .env file:
   - GROQ_API_KEY=your_groq_key
   - GROQ_API_ENDPOINT=https://api.groq.com/openai/v1
   - GROQ_MODEL=llama-3.1-70b-versatile (or another model)
3. **(Optional) Setup Langfuse for observability:** To trace agent execution and monitor performance, create a free account at https://langfuse.com and add these credentials to your .env file (optional, but recommended for debugging):
   - LANGFUSE_PUBLIC_KEY=your_public_key
   - LANGFUSE_SECRET_KEY=your_secret_key
   - LANGFUSE_HOST=https://cloud.langfuse.com (or your self-hosted instance)
4. **Set up the environment:** Create a virtual environment using uv venv and install dependencies using uv sync –frozen.

5. **Read this lab statement:** Review the entire lab document beforehand and take note of anything unclear to ask the instructors at the beginning of the session.

4. Deliverables

Submit the following through Moodle one week after the session:

- **Report (PDF):** Includes the developed code, justified answers to all questions, and analysis of results. The report should be concise but complete.

5. Theoretical Foundation: What is an AI Agent? (0.5 points)

What is an AI Agent?

An **AI agent** is an autonomous system that perceives its environment, reasons about goals, and takes actions to achieve them. Unlike a simple chatbot that produces a single response, an agent operates in a loop: it observes, decides, acts, and then observes the results of its actions before deciding again.

More formally, an AI agent is a program that:

1. Receives instructions or goals from the user.
2. Calls an AI model to reason about what action to take next.
3. Optionally invokes **tools** (external functions) to gather information or affect the world.
4. Processes tool results, updates its internal state, and repeats until it can provide a final answer.

The key distinction from traditional LLM applications is *autonomy*: the agent decides when to stop, which tools to use, and how to combine information from multiple sources—all without explicit step-by-step instructions from the user.

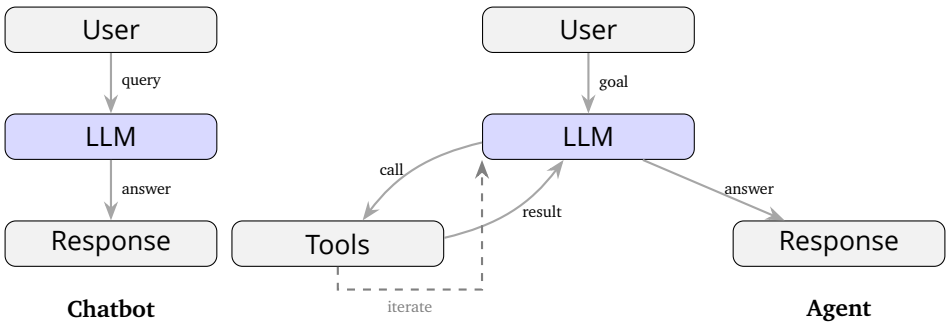

Figure 1. Chatbot vs. Agent: A chatbot produces a single response, while an agent iteratively uses tools to accomplish goals..

The ReAct Loop

The most common agent architecture is the **ReAct** (Reasoning + Acting) pattern. In this approach, the model alternates between reasoning about what to do and taking actions via tool calls. The loop continues until the model determines it has enough information to answer the user's query.

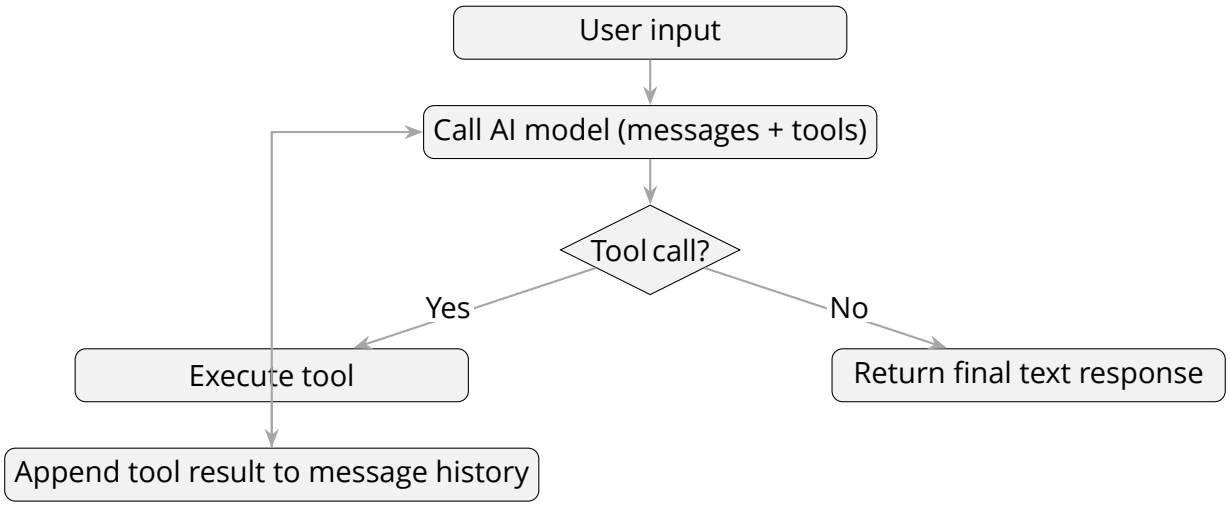

Figure 2. The ReAct agent loop: the model reasons, optionally calls tools, and repeats until it produces a final response..

This loop is deceptively simple but remarkably powerful. The model uses tools to extend its capabilities beyond pure text generation—retrieving real-time data, performing calculations, querying databases, or interacting with external APIs.

6. Tools and the Tool System (1.5 points)

In this lab, you will build your own minimal agent framework from scratch. This hands-on approach helps you understand exactly how agents work internally, without the abstraction layers that high-level frameworks introduce. We start by implementing the tool system, which is the foundation that allows agents to take actions.

What is a Tool?

A **tool** is a function that an agent can invoke to interact with the external world. Each tool has three components:

- A **name** and **description** so the model knows what the tool does and when to use it.
- A **parameter schema** (in JSON Schema format) so the model knows what arguments to pass.
- An **implementation** that executes the action and returns a result as a dictionary.

For example, an `add` tool would take two numbers as input and return their sum.

6.1. Step 1: Define the Tool Class

The foundation of our framework is a `Tool` class that represents a callable function. We use Python's `dataclass` decorator for a clean, minimal implementation. Since manually writing JSON Schema can be cumbersome, we will use **Pydantic BaseModel classes** for parameters and convert them to JSON Schema automatically:

```
from dataclasses import dataclass
from typing import Any, Callable, Type

from pydantic import BaseModel

@dataclass
```

```
class Tool:
    """Represents a tool that an agent can use."""
    name: str                                   # Unique identifier
    description: str                            # What it does
    parameters: Type[BaseModel]                 # Pydantic model class for parameters
    func: Callable[[dict[str, Any]], dict[str, Any]]  # The function to call
```

In addition to these attributes, our Tool class needs two methods: one to convert the tool into the format expected by the OpenAI API, and one to execute the tool.

6.1.1. Method 1: to_openai_format

In this lab we will use the openai Python library (as in previous labs), so our tool definitions must match the OpenAI tool-calling API schema. This method serializes our tool into that format (see the official documentation: https://platform.openai.com/docs/guides/function-calling). Since parameters is a Pydantic model class, we use model_json_schema() to generate the JSON Schema automatically:

```
def to_openai_format(self) -> dict[str, Any]:
    """Convert to OpenAI's tool format."""
    return {
        "type": "function",
        "function": {
            "name": self.name,
            "description": self.description,
            "parameters": self.parameters.model_json_schema(),
        },
    }
```

The model_json_schema() method automatically converts the Pydantic model into a valid JSON Schema dictionary, including all field types, descriptions, and required fields. This eliminates the need to manually write JSON Schema dictionaries as they can be quite cumbersome to write by hand.

6.1.2. Method 2: execute

This method calls the underlying function with the provided arguments and returns the result:

```
def execute(self, arguments: dict[str, Any]) -> dict[str, Any]:
    """Execute the tool with the given arguments."""
    return self.func(arguments)
```

6.2. Step 2: Create an Example Tool

To illustrate how to create a tool, we will build a simple calculator tool that performs basic arithmetic operations. Before diving in, let's briefly review Pydantic.

6.2.1. Understanding Pydantic

Pydantic is a Python library that lets you define validated data schemas using standard type annotations. Pydantic models act as smart containers for your data: they check that the structure and types are correct, and provide automatic conversion (for example, converting strings to numbers if possible).

To use Pydantic, you create classes that inherit from BaseModel. Each class variable becomes a validated field with a specified type. Pydantic uses these annotations to validate any data passed to the model, ensuring errors are caught early.

Here's what this looks like in practice:

```
from typing import Literal
from pydantic import BaseModel, Field


class CalculatorParams(BaseModel):
    """Defines the expected arguments for a calculator tool."""
    x: float = Field(description="First number to use in the operation")
    y: float = Field(description="Second number to use in the operation")
    operation: Literal["add", "subtract", "multiply", "divide"] = Field(
        description="The operation to perform. Options: add, subtract, multiply, divide"
    )


# Validate and parse user input
params = CalculatorParams(operation="add", x=2, y="3.5")
print(params)  # operation='add' x=2.0 y=3.5
```

Notice that Pydantic can coerce compatible datatypes (like converting the string "3.5" to the float 3.5) and will raise clear errors for invalid input.

The Field function adds metadata such as a description for each field. This information is included in the auto-generated JSON Schema, accessible via CalculatorParams.model_json_schema().

6.2.1.1. Adding examples to the schema.

For more complex tools, it is useful to include examples in the schema. This can be done using the model_config attribute:

```
class CalculatorParams(BaseModel):
    """Defines the expected arguments for a calculator tool."""
    x: float = Field(description="First number to use in the operation")
    y: float = Field(description="Second number to use in the operation")
    operation: Literal["add", "subtract", "multiply", "divide"] = Field(
        description="The operation to perform. Options: add, subtract, multiply, divide"
    )

    model_config = {
        "json_schema_extra": {
            "examples": [
                {"operation": "add", "x": 2, "y": 3},
                {"operation": "subtract", "x": 5, "y": 2},
                {"operation": "multiply", "x": 3, "y": 4},
                {"operation": "divide", "x": 10, "y": 2},
            ]
        }
    }
```

6.2.2. Defining the Tool

Now let's create a calculator tool to test our implementation. We'll reuse the CalculatorParams model we defined earlier, and use it to validate the input arguments:

```
"""
Calculator tools for AI agents.

Provides basic arithmetic operations as tools that agents can use.
```

```
"""
def calculator(args: dict[str, Any]) -> dict[str, Any]:
    """Perform basic arithmetic operations."""
    try:
        params = CalculatorParams(**args)
        if params.operation == "add":
            return {"result": params.x + params.y}
        elif params.operation == "subtract":
            return {"result": params.x - params.y}
        elif params.operation == "multiply":
            return {"result": params.x * params.y}
        elif params.operation == "divide":
            return {"result": params.x / params.y}
        else:
            return {"error": "Invalid operation"}
    except Exception as e:
        return {"error": f"Error performing operation: {str(e)}"}


# Create tools
calculator_tool = Tool(
    name="calculator",
    description="Perform basic arithmetic operations",
    parameters=CalculatorParams,
    func=calculator,
)

__all__ = ["calculator_tool"]
```

Exercise 6.1: Build a Weather Tool.

You will implement a **weather** tool that returns current conditions for a city. Use the following Open-Meteo APIs. Implement a Pydantic params model, the tool function, and register it as a Tool in lab4_agents/tools/weather.py.

Geocoding API. Convert a location name into geographic coordinates and a display name.

Example request for Madrid: https://geocoding-api.open-meteo.com/v1/search?name=Madrid&count=1

Forecast API. Retrieve the current weather conditions for specific coordinates.

Example request for Madrid: https://api.open-meteo.com/v1/forecast?latitude=40.4165&longitude=-3.70256&current=temperature_2m,apparent_temperature,relative_humidity_2m,wind_speed_10m,wind_gusts_10m,weather_code

Your tool should accept a single parameter (e.g. location: a string), call the geocoding API, then the forecast API, and return a dictionary with the relevant fields (and handle errors, e.g. location not found or network failure). Later you will use this tool in a multi-tool agent to **compare the temperature of two cities**.

7. Creating Agents (3 points)

Now that we have a working tool system, we can build the agent itself. The agent is responsible for orchestrating the conversation: it receives user input, calls the language model, executes tools when requested, and returns the final response.

7.1. Step 1: API Configuration

Before building the agent, we need to configure the connection to our language model. We use the GROQ API, which is OpenAI-compatible, meaning we can use the standard openai Python library:

```
import os
from dotenv import load_dotenv
from openai import OpenAI

load_dotenv()

GROQ_API_ENDPOINT = os.getenv("GROQ_API_ENDPOINT")
GROQ_API_KEY = os.getenv("GROQ_API_KEY")
GROQ_MODEL = os.getenv("GROQ_MODEL", "llama-3.1-70b-versatile")

client = OpenAI(api_key=GROQ_API_KEY, base_url=GROQ_API_ENDPOINT)
```

Observability with Langfuse

This lab builds on the observability setup from the previous lab. We continue using **Langfuse**, an open-source LLMOps platform that provides tracing, monitoring, and analytics for LLM applications.

Setup (same as previous lab): If you have already configured Langfuse credentials in your .env file from the previous lab, you can skip this step. The same credentials work here. If you want to keep this lab's traces separate from the previous lab, create a new Langfuse project in your dashboard and use its credentials.

The implementation uses the @observe decorator from Langfuse to instrument key methods:

```
from langfuse import observe

@observe(as_type=\"generation\")
def _call_api(self) -> ChatCompletionMessage:
    \"\"\"Make an API call with automatic Langfuse tracing.\"\"\"
    # ... implementation ...

@observe(as_type=\"span\")
def _execute_tool_call(self, tool_call):
    \"\"\"Execute a tool with automatic tracing.\"\"\"
    # ... implementation ...

@observe()
def run(self, user_input: str, max_iterations: int = 10) -> str:
    \"\"\"Main agent loop with automatic tracing.\"\"\"
    # ... implementation ...
```

For the sections that we tell you to look at the messages please use Langfuse.

7.2. Step 2: Agent Initialization

The Agent class stores everything needed to manage a conversation:

- A reference to the API client.
- The model identifier.
- Available tools (stored as a dictionary for fast lookup by name).

- An optional system prompt that defines the agent's behavior.
- The message history, which provides context across multiple turns.

```python
from typing import Any
from lab4_agents.tool import Tool
from lab4_agents.config import client, GROQ_MODEL


class Agent:
    def __init__(
        self,
        model: str | None = None,
        tools: list[Tool] | None = None,
        system_prompt: str | None = None,
    ):
        self.client = client
        self.model = model or GROQ_MODEL
        self.tools = {tool.name: tool for tool in (tools or [])}

        # Set default system prompt if none provided (best practice)
        if system_prompt is None:
            system_prompt = (
                "You are a helpful AI assistant with access to various tools. "
                "Use the available tools when necessary to answer user questions accurately. "
                "After receiving tool results, provide a clear final answer to the user. "
                "Do NOT call the same tool repeatedly with identical arguments. "
                "Think step by step and explain your reasoning."
            )

        self.system_prompt = system_prompt
        self.messages: list[dict[str, Any]] = []

        # Add system prompt to message history
        if system_prompt:
            self.messages.append({"role": "system", "content": system_prompt})
```

7.3. Step 3: Converting Tools to OpenAI Format

When calling the API, we must provide tools in OpenAI's expected format. This helper method converts all registered tools:

```python
def _get_openai_tools(self) -> list[dict[str, Any]] | None:
    """Get tools in OpenAI format."""
    if not self.tools:
        return None
    return [tool.to_openai_format() for tool in self.tools.values()]
```

If no tools are registered, we return None rather than an empty list. This tells the API that the agent has no tool-calling capability.

7.4. Step 4: Calling the API with Retry Logic

The _call_api method makes a single request to the language model with the current message history and available tools. To ensure robustness in production environments, this method includes **automatic retry logic** using exponential backoff:

```python
from openai.types.chat import ChatCompletionMessage
from tenacity import (
    retry,
    stop_after_attempt,
    wait_exponential,
    retry_if_exception_type,
)


@retry(
    stop=stop_after_attempt(3),
    wait=wait_exponential(multiplier=1, min=2, max=10),
    retry=retry_if_exception_type((Exception,)),
)
def _call_api(self) -> ChatCompletionMessage:
    """
    Make a single API call with automatic retry logic.

    Implements exponential backoff with the following behavior:
    - Retries up to 3 times on any exception
    - Initial wait: 2 seconds
    - Exponential backoff: wait increases exponentially
    - Maximum wait: 10 seconds between retries
    """
    response = self.client.chat.completions.create(
        model=self.model,
        messages=self.messages,
        tools=self._get_openai_tools(),
    )
    return response.choices[0].message
```

subsubsection*Why Retry Logic?

Network issues, rate limits, and temporary service outages are common in production. The @retry decorator from the **tenacity** library automatically retries failed API calls with exponential backoff:

- **Fault tolerance**: Transient errors (network timeouts, 429 rate limits) are automatically retried.
- **Exponential backoff**: Wait time increases after each failure (2s, then 4s, then 10s) to avoid overwhelming the service.
- **Max attempts**: After 3 attempts, the exception is raised to prevent infinite retries.

The returned message object may contain:

- A content field with text (the model's response).
- A tool_calls field with one or more tool invocations.
- Both, if the model wants to explain its reasoning while also calling tools.

7.5. Step 5: Executing Tool Calls

When the model requests a tool, we need to execute it and return the result. The _execute_tool_call method handles a single tool invocation:

```python
import json
from openai.types.chat import ChatCompletionMessageToolCall


def _execute_tool_call(
    self, tool_call: ChatCompletionMessageToolCall
) -> dict[str, Any]:
    """Execute a single tool call and return the result."""
    tool_name = tool_call.function.name
    arguments = json.loads(tool_call.function.arguments)

    tool = self.tools.get(tool_name)
    if tool is None:
        return {"error": f"Tool '{tool_name}' not found"}

    try:
        return tool.execute(arguments)
    except Exception as e:
        return {"error": str(e)}
```

This method includes error handling for two failure modes: the tool not existing (which can happen if the model hallucinates a tool name) and the tool raising an exception during execution.

7.6. Step 6: Processing Multiple Tool Calls

Modern language models can request multiple tools in a single response. The _process_tool_calls method handles this by:

1. Adding the assistant's message (including tool call requests) to the history.
2. Executing each requested tool.
3. Adding each tool's result to the history with the correct tool_call_id.

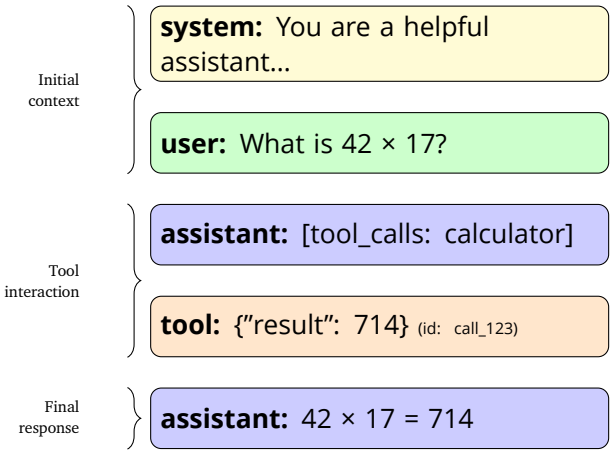

Figure 3. Message history structure: each interaction adds messages with specific roles, building context for the model..

```python
def _process_tool_calls(self, message: ChatCompletionMessage) -> None:
    """Process all tool calls in a message."""
    # Build and append the assistant message
    assistant_message: dict[str, Any] = {"role": "assistant"}
```

```
    if message.content:
        assistant_message["content"] = message.content

    if message.tool_calls:
        assistant_message["tool_calls"] = [
            {
                "id": tc.id,
                "type": tc.type,
                "function": {
                    "name": tc.function.name,
                    "arguments": tc.function.arguments,
                },
            }
            for tc in message.tool_calls
        ]

    self.messages.append(assistant_message)

    # Execute each tool and append results
    for tool_call in message.tool_calls:
        result = self._execute_tool_call(tool_call)
        self.messages.append({
            "role": "tool",
            "tool_call_id": tool_call.id,
            "content": json.dumps(result),
        })
```

The `tool_call_id` field is crucial: it links each result back to the specific tool call that produced it, allowing the model to correctly interpret results when multiple tools are called simultaneously.

7.7. Step 7: The Main Agent Loop with Iteration Limits

The `run` method implements the ReAct loop we discussed earlier. This is the core of the agent. To prevent infinite loops in edge cases, it includes an **iteration limit**:

```
def run(self, user_input: str, max_iterations: int = 10) -> str:
    """
    Run the agent with user input.
    """
    self.messages.append({"role": "user", "content": user_input})

    # Agent loop with iteration limit
    iteration = 0
    while iteration < max_iterations:
        iteration += 1

        # Call the API (with automatic retry logic)
        message = self._call_api()

        # Check if the model wants to use tools
        if message.tool_calls:
            self._process_tool_calls(message)
            continue  # Loop back to get the next response

        # No tool calls means we have a final response
        self.messages.append({"role": "assistant", "content": message.content})
```

```
        return message.content

    # Max iterations reached
    raise RuntimeError(
        f"Maximum iterations ({max_iterations}) reached. "
        "The agent may be stuck in a loop. Check your system prompt and tool implementations."
    )
```

subsubsection*Iteration Limits and Why They Matter

The loop continues until the model returns a message without any tool calls OR until max_iterations is reached. This safeguard prevents several failure modes:

- **Infinite loops**: If a tool always returns the same result and the model keeps calling it, the agent will eventually hit the limit.
- **Stuck states**: If the model's reasoning enters a cycle without reaching a conclusion, the limit stops it gracefully.
- **Cost control**: In production, you can set max_iterations to control API costs for runaway agents.
- **User experience**: Prevents agents from taking too long, improving responsiveness.

Choosing the right limit: The default of 10 iterations works for most tasks. For complex multi-step queries, increase it when calling run():

```
# Simple query - 10 iterations is usually enough
response = agent.run(\"What is 2 + 2?\")

# Complex multi-step query - may need more iterations
response = agent.run(
    \"Calculate the average of [1, 2, 3, 4, 5], then multiply by 2, \"\
    \"then check if the result is prime\",
    max_iterations=20
)
```

7.8. Step 8: Resetting the Conversation

For multi-turn conversations, it is useful to be able to clear the history while preserving the system prompt:

```
def reset(self) -> None:
    """Clear conversation history while keeping the system prompt."""
    self.messages = []
    if self.system_prompt:
        self.messages.append({"role": "system", "content": self.system_prompt})
```

7.9. Putting It All Together

With all components in place, we can now test our agent with the calculator tool:

```
from lab4_agents.agent import Agent
from lab4_agents.tools.calculator import calculator_tool

agent = Agent(
```

```
    tools=[calculator_tool],
    system_prompt="You are a helpful assistant. Use tools when needed.",
)

response = agent.run("What is 42 multiplied by 17?")
print(response)
```

When you run this, the agent will:

1. Receive the user's question.
2. Call the API, which recognizes the need for calculation.
3. Request the calculator tool with operation="multiply", x=42, y=17.
4. Execute the tool and receive {"result": 714}.
5. Call the API again with the tool result in context.
6. Return a natural language response like "42 multiplied by 17 equals 714."

Exercise 7.1: Use the provided Agent class in lab4_agents/agent.py. Test it with the calculator tool and verify that it correctly handles:

- Simple arithmetic questions (single tool call).
- Questions that do not require tools (direct response).
- Questions with invalid operations (error handling).

Include sample outputs in your report.

Question 7.1: Why does the agent loop continue after processing tool calls instead of returning immediately? What would happen if we returned the tool result directly to the user?

Question 7.2: Examine the message history after a successful tool-calling interaction. How many messages are added for a single tool call? Draw a diagram showing the message flow.

Production Considerations: Robustness and Observability

Building agents for production requires more than just a working loop. The implementation includes three critical features for reliability:

1. Automatic Retry with Exponential Backoff

Network issues happen. The @retry decorator ensures transient failures don't crash your agent:

```
# If a network timeout occurs, the agent automatically retries
# Retry attempt 1: wait 2 seconds
# Retry attempt 2: wait ~4 seconds
# Retry attempt 3: wait ~10 seconds
# After 3 attempts, the error is raised to the caller
agent.run("Tell me a joke")  # Survives temporary network issues
```

2. Iteration Limits Prevent Runaway Loops

Without iteration limits, a buggy tool or poor system prompt could cause infinite loops:

```
# Default: max 10 iterations (usually enough for simple tasks)
agent.run("What is 2 + 2?")

# Complex tasks may need more iterations
agent.run("Solve this puzzle", max_iterations=25)

# If max is hit, a clear error is raised (not a hang)
# RuntimeError: Maximum iterations (10) reached...
```

3. Langfuse Observability for Debugging

When things go wrong, Langfuse provides full trace visibility:

```
# With Langfuse credentials configured, every agent.run() is logged
# You can see in the Langfuse dashboard:
# - Message history and tool calls
# - Token usage and API costs
# - Latency for each step
# - Errors with full context

agent.run("What is the capital of France?")
# Check Langfuse dashboard → see the full trace
```

These three features work together: retries keep agents resilient, iteration limits prevent runaway agents, and Langfuse provides visibility when things don't work as expected. Together they make agents suitable for production use.

Exercise 7.2: Build a **multi-tool agent** that has both the calculator and the weather tool. Test it with:

- One query that uses only the calculator (e.g. "What is 42 times 17?").
- One query that uses only the weather tool (e.g. "What is the weather in Madrid?").
- One query that **compares the temperature of two cities** (e.g. "Which is warmer right now, Madrid or Barcelona?" or "What is the temperature in Madrid and in Barcelona?"). The agent should call the weather tool for each city and then answer.

Include the prompts and responses (or a short transcript) for these three runs in your report.

8. Advanced Agentic Patterns

So far, we have implemented a basic ReAct agent that can use tools to answer questions. However, real-world agents often need more sophisticated reasoning capabilities. This section introduces two advanced patterns that improve agent performance: **Chain of Thought (CoT) prompting** and **Self-Correction**.

8.1. Chain of Thought Prompting

Chain of Thought (CoT) prompting is a technique that encourages the language model to show its reasoning process step-by-step before producing a final answer. Instead of jumping directly to a conclusion, the model breaks down complex problems into intermediate reasoning steps.

Why Chain of Thought?

Standard prompting often leads models to produce answers without showing their reasoning. This makes it difficult to:

- Debug why an agent made a particular decision.
- Understand the agent's thought process.
- Improve the agent's performance through better prompting.

CoT prompting addresses these issues by explicitly asking the model to "think aloud," which often leads to more accurate results, especially for multi-step problems.

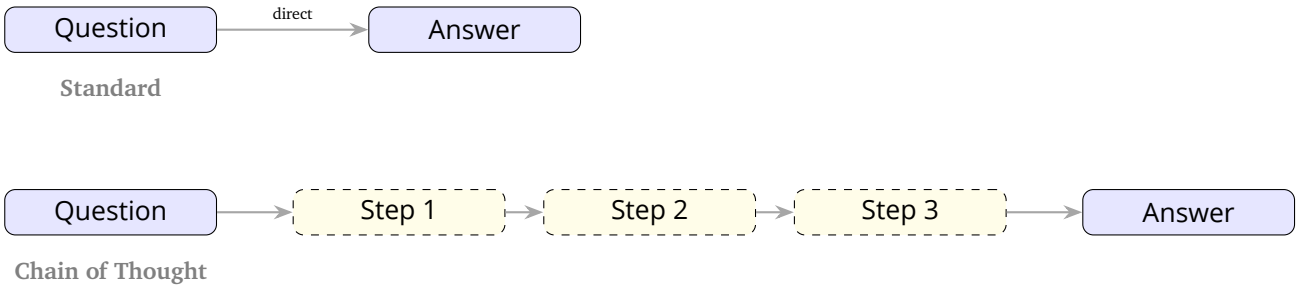

Figure 4. Standard vs. Chain of Thought: CoT makes the model show intermediate reasoning steps before the final answer..

Implementing Chain of Thought in Agents

To implement CoT in our agent, we modify the system prompt to encourage step-by-step reasoning. The agent should explicitly show its reasoning before deciding whether to use tools or provide a final answer.

```
agent = Agent(
    tools=[calculator_tool],
    system_prompt="""You are a helpful assistant that solves problems step-by-step.

When solving a problem:
1. First, think about what information you need.
2. Break down the problem into smaller steps.
3. Use tools when necessary to gather information or perform calculations.
4. Show your reasoning at each step.
5. Finally, provide a clear answer.

Always explain your reasoning before taking any action.""",
)
```

Example: Multi-Step Problem Solving

Consider a question like "What is the average of 10, 20, and 30?" Without CoT, the model might directly call the calculator. With CoT, it should reason:

1. *To find the average, I need to:*
    - Sum the numbers: $10 + 20 + 30 = 60$
    - Divide by the count: $60/3 = 20$

2. *I'll use the calculator tool to perform these operations.*

This explicit reasoning helps the model avoid errors and makes the agent's behavior more transparent.

Advanced CoT: Few-Shot Examples

For more complex reasoning, we can provide few-shot examples in the system prompt that demonstrate the desired reasoning pattern:

```
system_prompt = """You are a helpful assistant that solves problems step-by-step.

Example 1:
User: What is 15 plus 27?
Assistant: I need to add 15 and 27. Let me use the calculator tool.
[Tool call: calculator with operation="add", x=15, y=27]
Tool result: {"result": 42}
The answer is 42.

Example 2:
User: What is twice the sum of 5 and 8?
Assistant: I need to:
1. First find the sum: 5 + 8 = 13
2. Then multiply by 2: 13 * 2 = 26
Let me use the calculator tool for both steps.
[Tool call: calculator with operation="add", x=5, y=8]
Tool result: {"result": 13}
[Tool call: calculator with operation="multiply", x=13, y=2]
Tool result: {"result": 26}
The answer is 26.

Now solve the user's problem following this pattern."""
```

Exercise 8.1:[Try CoT on one run] Change your agent's system prompt to the CoT version shown in this lab (or a simplified variant). Run the agent on **one** multi-step arithmetic question (e.g. "What is the average of 10, 20, and 30?"). Copy the model's full reply (including any reasoning text) into the report and try to see if it is more resilient to errors than the standard version.

8.2. Self-Correction

Self-correction is a pattern where an agent verifies its own output and corrects errors when detected. This is particularly useful for tasks where accuracy is critical, such as mathematical calculations, code generation, or factual queries.

The Self-Correction Loop

A self-correcting agent follows this pattern:

1. Generate an initial answer or solution.
2. Verify the answer (using tools, validation logic, or a verification prompt).
3. If verification fails, analyze the error and generate a corrected answer.
4. Repeat until verification passes or a maximum number of attempts is reached.

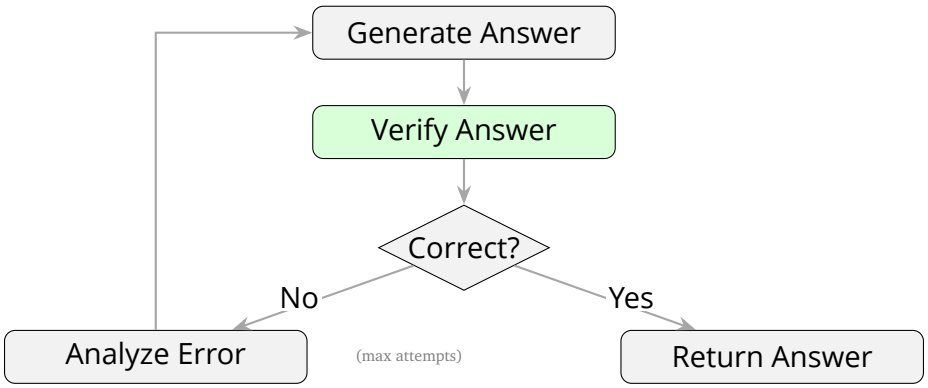

Figure 5. Self-correction loop: generate, verify, and correct until the answer passes verification or max attempts reached..

Implementing Self-Correction

We can extend our Agent class with a self-correction mechanism. The key is to add a verification step after the agent produces a response:

```
def run_with_self_correction(
    self,
    user_input: str,
    max_attempts: int = 3,
    verification_prompt: str | None = None
) -> str:
    """Run the agent with self-correction enabled."""
    default_verification = """Review your previous answer. Is it correct?
    If there are any errors, explain what went wrong and provide a corrected answer."""

    verification = verification_prompt or default_verification

    for attempt in range(max_attempts):
        # Get initial response
        response = self.run(user_input)

        # Verify the response
        self.messages.append({
            "role": "user",
            "content": verification
        })

        verification_response = self._call_api()

        # Check if verification indicates an error
        if "error" in verification_response.content.lower() or \
           "incorrect" in verification_response.content.lower() or \
           "wrong" in verification_response.content.lower():
            # Use the corrected answer from verification
            if attempt < max_attempts - 1:
                self.messages.append({
                    "role": "assistant",
                    "content": verification_response.content
                })
                continue  # Try again with correction

        # Verification passed or max attempts reached
```

```
        return response

    return response  # Return last attempt
```

Tool-Based Verification

For mathematical or computational tasks, we can use tools to verify answers automatically:

```
def verify_calculation(original_query: str, answer: str, calculator_tool: Tool) -> bool:
    """Verify a calculation answer by re-computing."""
    # Extract numbers and operation from the query (simplified example)
    # In practice, you might use regex or another LLM call to parse

    # Re-run the calculation
    # Compare results
    # Return True if they match, False otherwise
    pass
```

Example: Self-Correcting Calculator Agent

Here's a complete example of a self-correcting agent for mathematical problems:

```
agent = Agent(
    tools=[calculator_tool],
    system_prompt="""You are a mathematical assistant.
    Always show your work step-by-step.
    After providing an answer, verify it by re-checking your calculations.""",
)

# The agent will:
# 1. Solve the problem
# 2. Verify its answer
# 3. Correct if needed
response = agent.run_with_self_correction(
    "What is (15 + 25) * 2?",
    max_attempts=3
)
```

Benefits and Limitations

Self-correction improves accuracy but comes with trade-offs:

Benefits:

- Higher accuracy through error detection and correction.
- More reliable outputs for critical tasks.
- Better handling of edge cases.

Limitations:

- Increased API calls (and cost) due to verification steps.
- Slower response times.
- May not catch all errors if verification logic is flawed.
- Can lead to infinite loops if verification always fails.

To mitigate these limitations, always set a maximum number of correction attempts and carefully design verification logic.

Exercise 8.2: Extend your `Agent` class with a `run_with_self_correction` method.

- **Warm-up / trace:** Before or alongside implementation, run self-correction once on a query where verification might trigger (e.g. a multi-step calculation). In the report: note the **verification message** sent, the **model's verification response**, and whether a **correction step** occurred. This makes the loop concrete.
- **Concrete test cases:** Test with the following (at least 2–3):
  - **Easy:** "What is 17 + 23?" (baseline; unlikely to correct).
  - **Multi-step:** "What is (10+20+30)/3 + 5?" (may trigger verification or correction).
  - **Incomplete or ambiguous:** One prompt where the initial answer might be wrong or incomplete so self-correction can fix it.
- **Deliverable:** Provide a short comparison **with vs. without** self-correction: accuracy (correct/incorrect), number of API calls, and response time; plus **at least one example** where self-correction actually corrected an error (paste the before/after or describe it).
- **Optional:** Implement or sketch **tool-based verification** (e.g. use the calculator to re-check a numeric answer) and run self-correction with it once; compare to prompt-only verification. This ties to the "Tool-Based Verification" code stub in the lab.

Include your findings in the report.

Question 8.1: How would you implement self-correction for a code-generation agent? What verification mechanisms would you use?

Question 8.2: Discuss the trade-offs between CoT prompting and self-correction. When would you use one, both, or neither?

8.2.1. Orchestrator Agent

One powerful pattern in multi-agent systems is the **orchestrator agent**: a main agent that coordinates specialized sub-agents by using them as tools. This enables hierarchical agent architectures where each agent has a specific role, and a coordinator agent decides which specialized agent to invoke for different tasks.

8.2.1.1. Why Orchestrator Agents?

In complex systems, it's often beneficial to have:

- **Specialized agents** that excel at specific tasks (e.g., a calculator agent, a code analysis agent, a data retrieval agent).
- **An orchestrator agent** that understands the user's intent and delegates to the appropriate specialist.

This separation of concerns makes the system more modular, easier to maintain, and allows each agent to have focused capabilities and system prompts.

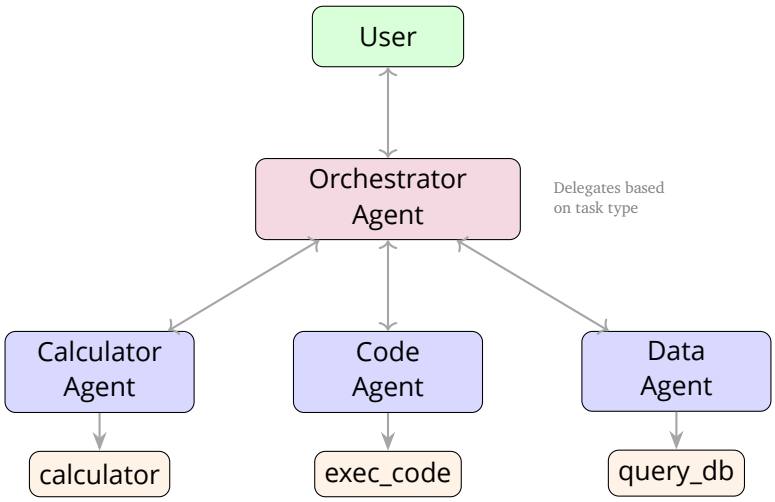

Figure 6. Orchestrator pattern: a central agent delegates to specialized sub-agents, each with their own tools..

8.2.1.2. Implementing Agent-as-Tool

To enable this pattern, we need a way to wrap an Agent instance as a Tool that can be used by other agents. We'll add a static method as_tool to the Agent class. Let's build this step by step.

8.2.1.2.1. Step 1: Define the Parameter Schema

First, we need to define what parameters the agent-tool will accept. Since we're wrapping an agent, the tool needs to accept the user input that will be passed to that agent. We use a Pydantic model for this:

```
from pydantic import BaseModel, Field

class AgentToolParams(BaseModel):
    """Parameters for calling an agent as a tool."""
    user_input: str = Field(
        description="The input/question to send to the agent. "
        "This will be processed by the agent using its tools and capabilities."
    )
```

This model ensures that when the orchestrator calls an agent-tool, it must provide a user_input string. The Field description helps the LLM understand what this parameter is for.

8.2.1.2.2. Step 2: Create the Method Signature

Now we add a static method to the Agent class. The method takes an agent instance and optional customization parameters:

```
class Agent:
    # ... existing code ...

    @staticmethod
    def as_tool(
        agent: "Agent",
        name: str | None = None,
        description: str | None = None
    ) -> Tool:
        """
        Create a Tool wrapper for an Agent instance.
```

```
        Args:
            agent: The Agent instance to wrap as a tool
            name: Optional custom name for the tool
                (defaults to agent's class name)
            description: Optional custom description
                        (defaults to a generic description)

        Returns:
            A Tool instance that wraps the agent
        """
```

The @staticmethod decorator means we can call this method without creating an Agent instance first: Agent.as_tool(calculator_agent).

8.2.1.2.3. Step 3: Handle Tool Naming and Description

Inside the method, we first determine the tool's name and description. If not provided, we generate sensible defaults:

```
tool_name = name or f"{agent.__class__.__name__.lower()}_agent"
tool_description = (
    description
    or f"Delegate tasks to a specialized {agent.__class__.__name__} agent. "
    f"Provide the user input/question, and the agent will process it "
    f"using its capabilities and tools."
)
```

For example, if you pass a CalculatorAgent instance without specifying a name, it will be called "calculatoragent_agent". The description helps the orchestrator understand when to use this tool.

8.2.1.2.4. Step 4: Create the Tool Execution Function

The core of the wrapper is a function that executes the agent when the tool is called. This function must match the signature expected by Tool: it takes a dictionary of arguments and returns a dictionary result:

```
def agent_tool_func(args: dict[str, Any]) -> dict[str, Any]:
    """Execute the agent with the provided user input."""
    try:
        params = AgentToolParams(**args)
        # Create a fresh instance to avoid state pollution
        fresh_agent = agent.__class__(
            model=agent.model,
            tools=list(agent.tools.values()),
            system_prompt=agent.system_prompt,
        )
        result = fresh_agent.run(params.user_input)
        return {"response": result}
    except Exception as e:
        return {"error": f"Error running agent: {str(e)}"}
    return Tool(
        name=tool_name,
        description=tool_description,
        parameters=AgentToolParams,
```

```
    func=agent_tool_func,
)
```

Key points:

- We validate the arguments using AgentToolParams(**args).
- We create a **fresh agent instance** using agent.__class__() to preserve the agent's type (important for subclasses). This ensures each tool call starts with a clean conversation history.
- We copy the original agent's configuration (model, tools, system prompt) to the new instance.
- We run the agent with params.user_input and wrap the result in a dictionary.
- Errors are caught and returned in a structured format.

Finally, we create and return a Tool instance that wraps everything together:

This Tool can now be added to another agent's tool list, allowing that agent to delegate tasks to the wrapped agent.

Note that this is a very simple implementation of the orchestrator agent. In a more complex implementation, we would need to handle the case where the agent is not able to handle the task, and we would need to retry the task with a different agent.

8.2.1.3. Example: Calculator Orchestrator

Let's create an orchestrator agent that uses a specialized calculator agent:

```
from lab4_agents import Agent
from lab4_agents.subagents import CalculatorAgent

# Create a specialized calculator agent
calculator_agent = CalculatorAgent()

# Wrap it as a tool
calculator_tool = Agent.as_tool(
    calculator_agent,
    name="calculator_agent",
    description="A specialized calculator agent that can perform "
                "arithmetic operations (addition, subtraction, "
                "multiplication, division). Use this when you need "
                "to perform mathematical calculations."
)

# Create an orchestrator agent that can delegate to the calculator
orchestrator = Agent(
    tools=[calculator_tool],
    system_prompt="""You are a helpful assistant that coordinates
    specialized agents to solve user problems. When a user asks a
    question that requires calculation, delegate it to the calculator
    agent. Always provide clear, natural language responses."""
)

# The orchestrator can now use the calculator agent
response = orchestrator.run(
    "I need to calculate the total cost: 15 items at $3.50 each, "
```

```
    "plus 8 items at $2.25 each. What's the total?"
)
```

When the orchestrator receives this query, it will:

1. Recognize that calculations are needed.
2. Call the calculator_agent tool multiple times (once for each multiplication, then for the addition).
3. Combine the results and provide a natural language answer.

8.2.1.4. Benefits of the Orchestrator Pattern

- **Modularity**: Each agent has a focused responsibility, making the system easier to understand and maintain.
- **Reusability**: Specialized agents can be reused across different orchestrators.
- **Scalability**: New capabilities can be added by creating new specialized agents without modifying existing ones.
- **Separation of concerns**: Each agent can have its own system prompt, tools, and configuration optimized for its specific task.
- **Composability**: Agents can be combined in different ways to create different orchestrator behaviors.

8.2.1.5. Limitations and Considerations

- **Latency**: Each agent invocation adds overhead, as it involves a full agent loop (potentially multiple API calls).
- **Cost**: Using agents as tools increases API calls, which can be expensive for high-volume applications.
- **State management**: Each agent invocation starts fresh, so agents cannot maintain context across multiple tool calls (this is by design to avoid state pollution).
- **Error propagation**: Errors in sub-agents need to be handled gracefully by the orchestrator.

Exercise 8.3: Implement the as_tool method in your Agent class. Create an **orchestrator** that has **two** specialized sub-agents as tools: one with the **calculator** only and one with the **unit converter** only (same tool as in exercises 1 and 3). The orchestrator must choose when to delegate to which agent.

- **Test cases:** (1) Single-step calculation. (2) Multi-step problems that require multiple calculator calls. (3) Questions that don't require calculations (the orchestrator should respond directly). (4) **Optional—combined query:** e.g. "Convert 100°F to Celsius and then add 10 to the result" (unit converter then calculator), so the orchestrator chains two different sub-agents.
- **Message flow:** For at least one multi-step query, provide an **ordered trace** in your report: (1) user message to orchestrator; (2) orchestrator's tool call(s) to sub-agent(s) with the user_input passed; (3) sub-agent's internal tool calls and results (if any); (4) sub-agent response back to orchestrator; (5) orchestrator's final response to the user. Optionally include a simple diagram.
- **Delegate vs. direct:** Give **one example** where the orchestrator correctly delegates to a specialized agent and **one example** where it answers directly without calling a tool; briefly explain why it chose each.

Include sample interactions in your report.

Question 8.3: How does the orchestrator pattern differ from simply giving the main agent all the tools directly? What are the trade-offs?

Question 8.4: Consider a scenario where you have three specialized agents: a calculator agent, a unit converter agent, and a currency converter agent. How would you design an orchestrator that can coordinate all three? What system prompt would you use?

8.2.2. Plan-and-Execute Agent

The **Plan-and-Execute** pattern is a powerful agent architecture that separates planning from execution. Instead of having a single agent reason and act in one loop, this pattern uses two distinct phases:

1. **Planning**: A planner agent breaks down complex tasks into a multi-step plan.
2. **Execution**: An executor agent executes each step sequentially, using tools as needed.
3. **Re-planning**: After execution, the planner evaluates whether the task is complete and generates follow-up plans if needed.

This architecture is based on research from Wang et al.'s Plan-and-Solve Prompting and Yohei Nakajima's BabyAGI project. It offers several advantages over the standard ReAct pattern.

8.2.2.1. Why Plan-and-Execute?

The standard ReAct agent calls the LLM for every tool invocation, which can be inefficient for complex, multi-step tasks. The Plan-and-Execute pattern addresses this by:

- **Reducing LLM calls**: The planner generates a complete plan upfront, reducing the number of reasoning calls needed during execution.
- **Better task decomposition**: Explicit planning helps break down complex tasks into clear, sequential steps.
- **Adaptive execution**: Re-planning allows the agent to adjust its strategy based on execution results.
- **Separation of concerns**: Planning and execution can use different models or prompts optimized for their specific roles.

However, this pattern still uses an LLM for each step execution and doesn't support variable assignment between steps, which limits its efficiency compared to more advanced architectures.

8.2.2.2. Architecture Overview

The Plan-and-Execute agent consists of three main components:

- **Planner**: An LLM that generates multi-step plans from user queries.
- **Executor**: A separate agent instance that executes individual plan steps using available tools.
- **Re-planning logic**: Evaluates task completion and generates follow-up plans when needed.

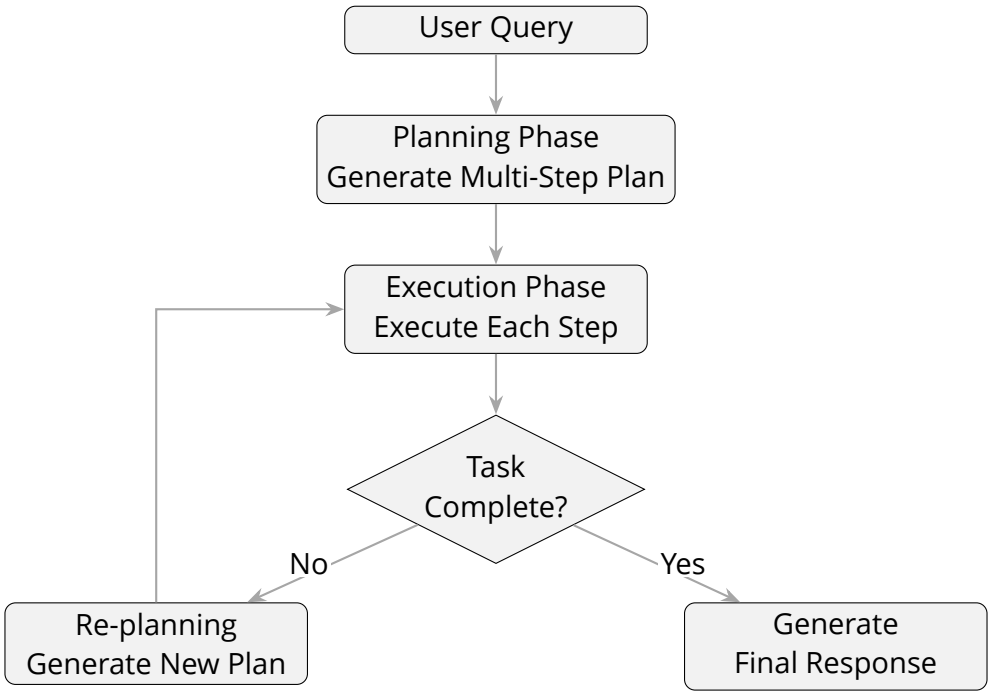

Figure 7. The Plan-and-Execute agent flow: plan generation, step execution, and adaptive re-planning..

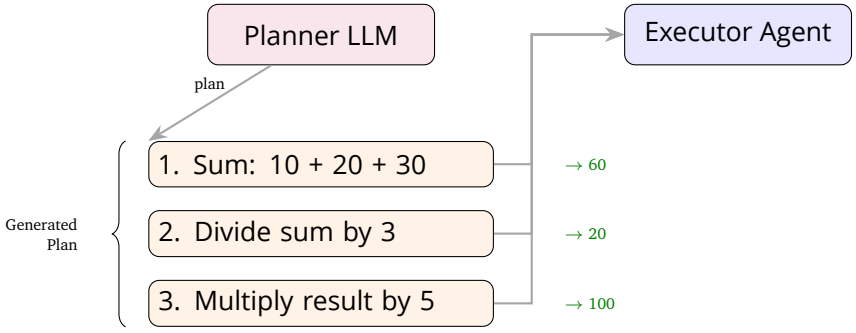

Figure 8. Plan-and-Execute example: planner generates steps, executor runs each with tools, results flow back..

8.2.2.3. Implementing the PlannerAgent

We'll create a PlannerAgent class that extends the base Agent class. The implementation includes methods for planning, execution, and re-planning.

8.2.2.3.1. Step 1: Initialization

The PlannerAgent initializes with separate configurations for planning and execution:

```
from lab4_agents.agent import Agent
from lab4_agents.tool import Tool

class PlannerAgent(Agent):
    """A Plan-and-Execute agent that breaks down complex tasks into steps."""

    def __init__(
        self,
        model: str | None = None,
        tools: list[Tool] | None = None,
        executor_model: str | None = None,
        executor_tools: list[Tool] | None = None,
```

```
    max_replan_iterations: int = 3,
    system_prompt: str | None = None,
):
    # Initialize base Agent for planning
    planner_prompt = (
        system_prompt
        or "You are a planning assistant. Break down complex tasks "
        "into clear, sequential steps. Generate a numbered list of "
        "actionable steps. Each step should be specific and executable."
    )
    super().__init__(
        model=model,
        tools=tools or [],
        system_prompt=planner_prompt,
    )

    # Create executor agent
    executor_model = executor_model or self.model
    executor_tools = executor_tools or tools or []
    executor_prompt = (
        "You are an execution assistant. Execute the given step using "
        "available tools. Be precise and return clear results."
    )
    self.executor = Agent(
        model=executor_model,
        tools=executor_tools,
        system_prompt=executor_prompt,
    )

    self.max_replan_iterations = max_replan_iterations
```

Key design decisions:

- The planner and executor can use different models (e.g., a larger model for planning, a smaller one for execution).
- Each has its own system prompt optimized for its role.
- The executor is a separate `Agent` instance to maintain clean separation.

8.2.2.3.2. Step 2: Plan Generation

The `plan` method generates a multi-step plan from a user query:

```
def plan(self, user_query: str) -> list[str]:
    """Generate a multi-step plan for the given query."""
    planning_prompt = (
        f"Break down the following task into clear, sequential steps.\n\n"
        f"Task: {user_query}\n\n"
        f"Generate a numbered list of steps (1., 2., 3., etc.). "
        f"Each step should be a single, actionable instruction."
    )

    # Use a temporary agent to avoid polluting conversation history
    temp_agent = Agent(
        model=self.model,
        tools=self.tools,
        system_prompt=self.system_prompt,
```

```
    )
    response = temp_agent.run(planning_prompt)

    # Parse the response to extract plan steps
    steps = self._parse_plan(response)
    return steps
```

The method uses a temporary agent instance to keep the planning conversation separate from the main execution flow. The response is then parsed to extract individual steps.

8.2.2.3.3. Step 3: Plan Parsing

The _parse_plan method extracts numbered steps from the LLM's text response:

```
import re

def _parse_plan(self, plan_text: str) -> list[str]:
    """Parse plan text into a list of step strings."""
    steps = []
    # Look for numbered list patterns: "1. Step", "2. Step", etc.
    patterns = [
        r"^\d+\.\s+(.+)$",      # "1. Step description"
        r"^\d+\)\s+(.+)$",      # "1) Step description"
        r"^Step\s+\d+[:\-]\s*(.+)$",  # "Step 1: description"
    ]

    lines = plan_text.strip().split("\n")
    for line in lines:
        line = line.strip()
        if not line:
            continue

        for pattern in patterns:
            match = re.match(pattern, line, re.IGNORECASE)
            if match:
                step = match.group(1).strip()
                if step:
                    steps.append(step)
                break

    # If no steps parsed, return original text as single step
    if not steps:
        steps = [plan_text.strip()]

    return steps
```

This parsing handles various numbering formats that LLMs might use, making the implementation robust to different response styles.

8.2.2.3.4. Step 4: Step Execution

The execute_step method executes a single plan step using the executor agent:

```
def execute_step(
    self,
    step: str,
    context: dict[str, Any] | None = None
```

```
) -> dict[str, Any]:
    """Execute a single plan step using the executor agent."""
    # Build execution prompt with context if available
    if context:
        context_str = json.dumps(context, indent=2)
        execution_prompt = (
            f"Execute the following step:\n\n{step}\n\n"
            f"Context from previous steps:\n{context_str}\n\n"
            f"Use the available tools to complete this step."
        )
    else:
        execution_prompt = (
            f"Execute the following step:\n\n{step}\n\n"
            f"Use the available tools to complete this step."
        )

    try:
        # Reset executor to avoid state pollution between steps
        self.executor.reset()
        result = self.executor.run(execution_prompt)
        return {"status": "success", "output": result}
    except Exception as e:
        return {"status": "error", "output": None, "error": str(e)}
```

The executor is reset before each step to ensure clean execution. Context from previous steps can be passed to help the executor understand the current state.

8.2.2.3.5. Step 5: Re-planning

After executing all steps, the agent evaluates whether the task is complete:

```
def replan(
    self,
    user_query: str,
    executed_steps: list[dict[str, Any]],
    current_plan: list[str],
) -> list[str] | None:
    """Evaluate if task is complete and generate new plan if needed."""
    # Build summary of executed steps
    step_summary = []
    for i, step_result in enumerate(executed_steps):
        step_num = i + 1
        status = step_result.get("status", "unknown")
        output = step_result.get("output", "")
        error = step_result.get("error", "")
        step_summary.append(
            f"Step {step_num}: {status}\n"
            f"Output: {output}\n"
            f"{f'Error: {error}' if error else ''}"
        )

    summary_text = "\n\n".join(step_summary)

    replanning_prompt = (
        f"Original task: {user_query}\n\n"
        f"Plan that was executed:\n"
        f"{chr(10).join(f'{i+1}. {step}' for i, step in enumerate(current_plan))}\n\n"
        f"Execution results:\n{summary_text}\n\n"
```

```
        f"Evaluate whether the task has been completed successfully. "
        f"If the task is complete, respond with 'TASK_COMPLETE'. "
        f"If additional steps are needed, generate a new numbered list of steps."
    )

    temp_agent = Agent(
        model=self.model,
        tools=self.tools,
        system_prompt=self.system_prompt,
    )
    response = temp_agent.run(replanning_prompt)

    # Check if task is complete
    if "TASK_COMPLETE" in response.upper():
        return None

    # Parse new plan if provided
    new_steps = self._parse_plan(response)
    if new_steps:
        return new_steps

    return None  # Safety fallback
```

The re-planning logic analyzes execution results and decides whether to continue or finish. This allows the agent to adapt when the initial plan doesn't fully solve the problem.

8.2.2.3.6. Step 6: Main Execution Loop

The run method orchestrates the entire Plan-and-Execute process:

```
def run(self, user_query: str) -> str:
    """Main entry point: Plan, execute, and re-plan as needed."""
    # Generate initial plan
    plan = self.plan(user_query)
    if not plan:
        return "Error: Could not generate a plan for the task."

    replan_count = 0
    all_executed_steps: list[dict[str, Any]] = []

    while replan_count <= self.max_replan_iterations:
        # Execute all steps in the current plan
        executed_steps: list[dict[str, Any]] = []
        context: dict[str, Any] = {}

        for step in plan:
            step_result = self.execute_step(step, context)
            executed_steps.append(step_result)

            # Update context with step result
            if step_result.get("status") == "success":
                context[f"step_{len(executed_steps)}"] = step_result.get("output", "")

        # Add executed steps to overall history
        all_executed_steps.extend(executed_steps)

        # Check if we need to re-plan
        if replan_count < self.max_replan_iterations:
```

```
            new_plan = self.replan(user_query, executed_steps, plan)
            if new_plan is None:
                # Task is complete, generate final response
                break
            # Update plan and continue
            plan = new_plan
            replan_count += 1
        else:
            # Max replan iterations reached
            break

    # Generate final response summarizing the results
    final_prompt = (
        f"Original task: {user_query}\n\n"
        f"All executed steps and their results:\n"
        f"{json.dumps(all_executed_steps, indent=2)}\n\n"
        f"Provide a clear, concise final answer to the user's task."
    )

    self.executor.reset()
    final_response = self.executor.run(final_prompt)
    return final_response
```

The main loop:

1. Generates an initial plan.
2. Executes all steps sequentially, building context along the way.
3. Evaluates completion and re-plans if needed (up to max_replan_iterations).
4. Generates a final response summarizing all results.

8.2.2.4. Example Usage

Here's a complete example using the PlannerAgent with calculator tools:

```
from lab4_agents.subagents import PlannerAgent
from lab4_agents.tools.calculator import CALCULATOR_TOOLS

# Create a planner agent with calculator tools
planner = PlannerAgent(
    tools=CALCULATOR_TOOLS,
    executor_tools=CALCULATOR_TOOLS,
    max_replan_iterations=3
)

# Execute a complex multi-step task
response = planner.run(
    "Calculate the average of 10, 20, and 30, then multiply the result by 5"
)
print(response)
```

The agent will:

1. Generate a plan like:
    - Step 1: Add 10, 20, and 30 to get the sum

- Step 2: Divide the sum by 3 to get the average
- Step 3: Multiply the average by 5

2. Execute each step using the calculator tool.
3. Verify completion and generate a final response.

8.2.2.5. Benefits and Limitations

Benefits:

- **Better task decomposition**: Explicit planning helps break down complex tasks.
- **Reduced planning overhead**: Plan once, execute many steps.
- **Adaptive execution**: Re-planning allows adjustment based on results.
- **Modular design**: Planner and executor can be optimized independently.

Limitations:

- **Still uses LLM per step**: Each step execution requires an LLM call, limiting efficiency.
- **No variable assignment**: Steps cannot directly pass variables to each other (context is passed as text).
- **Serial execution**: Steps execute sequentially, not in parallel.
- **Plan quality dependency**: Success depends on the planner generating good initial plans.

Exercise 8.4: Implement the PlannerAgent class following the design described above.

- **Test cases:** Use at least these three types:
  - **Single-step:** e.g. "What is 15 + 27?" (plan should have one step; executor uses calculator).
  - **Multi-step:** e.g. "Calculate the average of 10, 20, and 30, then multiply the result by 5" (clear plan of 3 steps).
  - **Re-planning:** Use a task where the first plan might be wrong or incomplete (e.g. a formulation that often leads to a failed step or an extra needed step), so you can observe and report re-planning.
- **Trace one run:** For one multi-step task, include in your report: (a) the **generated plan** (numbered steps as produced by the planner); (b) the **result of each step execution** (tool calls and outputs); (c) whether **re-planning** occurred and, if so, the new plan and why it was triggered. This makes plan → execution → (optional) replan concrete.
- **Comparison with ReAct:** For the same 2–3 tasks, provide a short **table**: ReAct agent vs. PlannerAgent, with columns such as: task; number of API calls (ReAct vs. Plan-and-Execute); success (yes/no); and a one-sentence observation (e.g. when Plan-and-Execute used fewer or more calls, or when re-planning helped).
- **Optional:** When would you use Plan-and-Execute vs. ReAct? Add a brief comment in the report.

Include your findings in the report.

Question 8.5: How does the Plan-and-Execute pattern handle cases where a step fails? What happens to subsequent steps in the plan?

Question 8.6: Consider a task like ”Search for information about X, then summarize it, then answer question Y based on the summary.” How would the Plan-and-Execute agent handle this differently from a ReAct agent? What are the trade-offs?

Question 8.7: The current implementation passes context between steps as text. How could you modify it to support structured variable passing? What would be the benefits and challenges?